Test Scenario Generation and Prioritization from UML 2.x Sequence Diagrams

Saroj Kanta Misra

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India

Test Scenario Generation and Prioritization from UML 2.x Sequence Diagrams

Thesis submitted in
May 2014
to the department of
Computer Science and Engineering
of
National Institute of Technology Rourkela
in partial fulfillment of the requirements
for the degree of

Master of Technology

in

Computer Science and Engineering

Specialization : Software Engineering

by

Saroj Kanta Misra
[Roll No. 212cs3470]

under the guidance of

Prof. Durga Prasad Mohapatra

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, India. www.nitrkl.ac.in

Mr. Durga Prasad Mohapatra
Associate Professor

Certificate

This is to certify that the work in the thesis entitled *Test Scenario Generation and Prioritization from UML 2.x Sequence Diagrams* by *Saroj Kanta Misra* is a record of an original work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of *Master of Technology* in *Computer Science and Engineering*. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Durga Prasad Mohapatra

Acknowledgment

I would like to express my gratitude to my thesis guide Prof. Durga Prasad Mohapatra for the useful comments, remarks and engagement through the learning process of this master thesis. The flexibility of work he has offered me has deeply encouraged me producing the research.

Furthermore I would like to thank Prof. Santanu Kumar Rath, Prof. Ashok Kumar Turuk, Prof. Banshidhar Majhi for for being a source of support and motivation for carrying out quality work.

Also, I like to thank the participants in my survey, who have willingly shared their precious time during the process of interviewing. I would like to thank my loved ones, who have supported me throughout entire process, both by keeping me harmonious and helping me putting pieces together. I will be grateful forever for your love.

Last but not the least,i like to thank my family and the one above all of us, the omnipresent God, for answering my prayers for giving me the strength to plod on despite my constitution wanting to give up, thank you so much Dear Lord.

Saroj Kanta Misra

Abstract

As the world is running towards greater heights of technology, it's becoming more complex to secure data from being copied. So it's better to detect the copied contents rather than securing the contents. Here, contents cover digital documents of scientific research, articles in newspapers, journals and assignments submitted by students. There are so many tools and algorithms to detect plagiarism, but the time complexity of the algorithm really matters where document comparison is against giant data set. Here we are proposing a new algorithm, which is developed using the concepts of the Relative Frequency Model. We also developed a tool based on our new approach which can check the similarity between test document and collection of registered documents. In this tool we implemented sentence splitter, and stop-word removal processes. And we will enhance our tool with various kinds of textual features like Stemming, Synonym Replacement, Word Sense Disambiguation and Web Crawler which is used for Web based Documents.

Keywords: Text Mining, Text Summarization, Stemming, Helmholtz Peinciple, Information Retrieval, Keyword Extraction, Term Frequency - Inverse Document Frequency.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, due to rapid increase in size and complexity of software applications, more emphasis is given towards object-oriented design strategy, which helps to reduce software cost and increase software reliability, and usability. But, introduction of object-oriented design and implementation approach brings out some new difficulties for software testing. Several features of object-oriented approach like polymorphism, dynamic binding, inheritance etc. create certain difficulties in software testing process. To test such object-oriented software from their implementation code is a very a complex process due to the different features of object oriented approach. Model Based Testing of these object oriented software can be beneficial to detect the error in the design phase itself, so that these error do not propagate to other stages of software development life cycle.

Control Flow Analysis (CFA) plays a vital role in determining all possible alternative paths a program may follow during execution. A Control Flow Graph (CFG) is a static representation of a program that represents all alternatives of control flow. For example, both choices for If ?? else statement can be represented in CFG as different control flow paths. A Loop can be represented as a cycle in a CFG.

According to Garousi et al. [1] Control flow information can be derived from two different sources: from software design artifacts and code itself. In Code-based CFA(CBCFA), control flow information is obtained from the available source code, whereas in Model-based CFA(MBCFA), control from information is obtained from design models such as UML.

1.1 Motivation

Any test case generation technique aims at generating effective test cases that can be useful for different applications. Now a days, object-oriented approach is fallowed for developing most of the applications, and these

object-oriented programs create problems for testing process because of its large size and complexity. Development time is wasted if we have implemented some faulty design, because if the design faults are detected after the code is written then we have to change both the design and the code.

Model Based Control Flow Analysis provides higher level of abstraction compared to Code Based Control Flow Analysis, So it is easier to extract control flow information in case of MBCFA, which is beneficial for testing process. The motivation of our work is to derive control flow information and generate test cases in the early stage of software development life cycle, after the UML design models of a system become available.

It is not practical to test a software exhaustively using each value that the input data may assume. In traditional testing techniques, each element of the software product is tested with equal through. Test case prioritization aims at finding an ordering of the test cases such that execution of the test cases in that order meets a given criterion. So, the aim of test case prioritization is to identify critical parts of software, for which more exhaustive testing is to be carried out. Test case prioritization is a well-known and efficient technique to ensure the software quality. Prioritization of test cases help in early detection of bugs and hence improves the quality of software. It provides a way of testing to achieve certain goals at faster rate.[9,17]

1.2 Objective

Based on the issues identified in the previous section, our objective is to focus on the behavioral aspects of the software. Different UML diagrams are used to represent the behavioral aspects of the system such as Interaction diagrams, Activity diagram , State machine diagram etc. Each of these diagrams used to represent the distinct behavior of the system, Hence test cases generated from these different diagrams will help to detect distinct type of faults in a complimentary way. So we set followings as our objective:

1. To develop a technique for effective and optimized test scenario generation using UML 2.x behavioral diagrams such
    - Interaction Diagram
    - Activity Diagram
    - State Machine Diagram
2. Prioritize test scenarios generated from using UML 2.x behavioral models

3. To develop an approach for generating and prioritizing test scenario from a combination of the UML 2.x behavioral diagrams.

4. Implement our proposed approach.

5. To make a comparison between the performance of the proposed approaches and related existing approaches.

Chapter 2

Basic Concepts and Definitions

In this section, we discuss some of the basic concepts and terminology associated with our proposed work.

2.1 Software Testing :

Software testing is the act of finding errors in a developed software. In other words, *"Testing is the process of executing a program with the intent of finding errors "*. Testing is an integral part of the software development process. As the systems becoming more and more software intensive, the need for better testing techniques continuously increased.

Software testing plays an important role in ensuring software quality. But exhaustive testing of large software is impractical. Increased size and complexity of software require better methods of testing activities in the software development life cycle. Further, need of rapid development forces software industries to develop quality software within aggressively smaller durations. In other words, developers have minimal lead time to assure the quality of software. Software testing attempts to detect all cases of incorrect behavior of a software product.

Software testing also measures quality of the software in terms its capability for achieving reliability, usability, correctness, maintainability, re-usability, and testability.

2.2 Testing Techniques :

There mainly three approaches for software testing, these are :

- Black Box Testing.
- White Box testing.

- Gray Box Testing.

In black box testing test cases are designed using only functional specification of the software, without any knowledge of the internal structure of the software. In other words, the structure or logic of the system is not considered. Hence in black box testinf test cases are designed based on the fucntional specifications. Input test data is given to the system, which checked against expected output after executing the software. Hence the tester have to design test cases for every possible set of inputs and outputs, which is a impractical job for the tester to do.

In white box testing the tester requires knowledge about the internal structure of software. The entire design, structure and code of the software have to be studied for this type of testing but it does not address the question of whether or not a program matches the specification and you if all of the functionality has been implemented.

Grey box testing is another type of software testing, which is a combination of black box and white box testing. Intention of this testing is to find out defects related to bad design or bad implementation of the system. In grey box testing, the test engineer is equipped with the knowledge of a system, and designs test cases or test data based on system knowledge.

Gray box testing or Model-Based Testing is a variant of testing that relies on explicit behavior models that encode the intended behavior of a system. Pairs of input and output of the model of the implementation are interpreted as test cases for this implementation: the output of the model is the expected output of the system under test (SUT).

2.3 Test Case :

The test case is a well-documented procedure designed to test the functionality of a feature in a system. The primary goal of designing test case is to detect error in a system. Basically a test case is the triplet [I, S, O], where I is the initial state of the system at which the test data is input, S is the state of the system at which the data will be input and O is the expected output of the system

2.4 Test Scenario :

A set of test cases that ensure that the business process flows are tested from end to end. They may be independent tests or a series of tests that follow each other, each dependent on the output of the previous one.

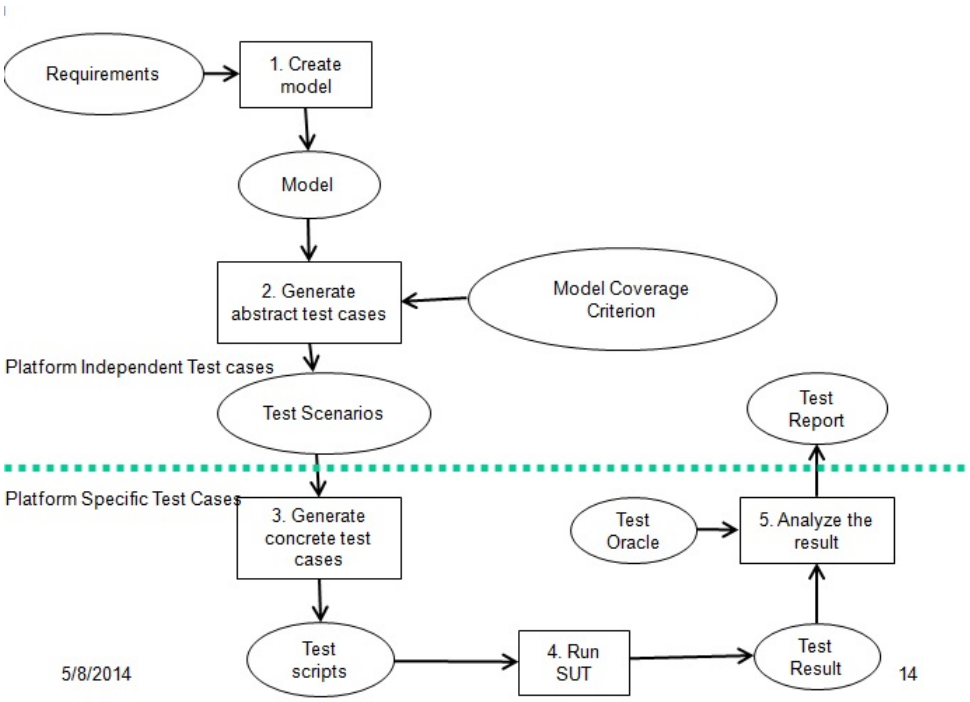


Figure 2.1: Model Based Testing approach

2.5 Overview of UML 2.0 :

There are thirteen different tyape of diagrams defined in UML 2.0. All these diagrams can be divided into three categories : Diagrams that represent the Static apllication structure of the system, Diagram represent general types of behavior and diagrams that represent dfferent aspects of interactions.

1. *Structure Diagram :* There are six different UML 2.0 diagrams that represent the static structure of the system, these are Class Diagram, Object Diagram, Composite Structure Diagram, Deployment Diagram, Object Diagram and Package Diagram.

2. *Behavior Diagram :* Three different UML 2.0 diagrams are used to represent the behavioral aspects of the system, these are Use Case Diagram (used by some methodologies during requirements gathering); Activity Diagram, and State Machine Diagram.

3. *Interaction Diagrams :* Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram includes in Interaction Diagrams.

2.6 Interaction Diagram :

Among all the UML 2.0 diagrams, the Interaction diagram describes the message-level details of an application, Which can be used for Control Flow Analysis of the system. Interaction diagrams describe how a group of objects collaborate in some behavior - typically a single use-case. Interaction diagrams are of two types: Sequence diagram and Communication diagram. The basic objective of both the diagrams are same. Sequence diagrams accentuate on the time sequence of messages passed between the communicating objects, and the communication diagrams accentuate on the structural organization of the communicating objects that send and receive messages. In our approach, we have used UML 2.x Sequence diagrams to generate test scenarios.

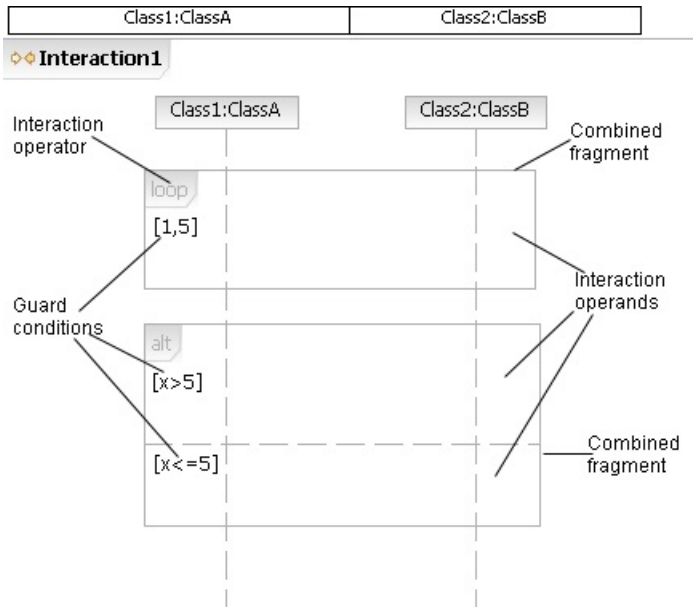


Figure 2.2: UML 2.0 Sequence Diagram with combined fragments

List below describes some of the feature of UML 2.O sequence diagrams

- **Interaction:** Series of messages that is passed between different communicating objects to satisfy some task is called interaction.
- **Interaction Occurrences:** When an interaction used within another interaction or context, then it is called interaction occurrence.
- **Combined Fragments:** Combined fragment is an interaction fragment which is a combination of multiple interaction fragments. Each combined fragment has an interaction operator and corresponding interaction operands.

- **Interaction Operand:** Interaction operand shows grouping of interactions within the combined fragment.
- **Interaction Operator:** Interaction operator for a combined fragments describes how the interaction operands present inside the combined fragments are going to be used. The followings are the list of interaction operators defined.
  - **Alternative (alt):** The interaction operator *alt* works like if-then-else structure. At most one operand can be selected based on the guard expression's true value. If there is no guard expression, then an implicit true guard value is implied.
  - **Option (opt):** The interaction operator *opt* is used, when the combined fragment represents the operand as an option where the operand either happens or may not happen. It works like an alternative combined fragment where one operand is nonempty and the other one is empty.
  - **Loop (loop):** The interaction operator loop represents a *loop* structure. The interaction operand present inside the loop combined fragment will be repeated many times. The repetition of loop can be controlled either or both by iteration bound and guard. If a loop combined fragment has no bound specified, then the loop will execute with infinite as upper bound and zero as lower bound.
  - **Break (break):** The interaction operator *break* is used to represent a breaking or exceptional scenario to be performed instead of the remaining interaction fragment.
  - **Parallel (par):** The interaction operator *par* describes parallel execution of behaviors of the interaction operands present inside a combined fragment.

2.7 Test Coverage Criteria For Sequence Diagram:

To design test cases a certain set of rules are fallowed, these rules guide which are the elements to be covered for testing. Here we discuss some of the test coverage criteria.

- **All-Messages-Criterion :** According to this criterion each message passed between two objects must present at least once in an adequate test. So this criterion ensures that every message that are passed between two object are covered by the test.
- **All-Path Coverage Criteria :** A set of test paths P, is said to satisfy all path coverage criteria, If it contain all the start-to-end message paths in a sequence diagram.

- **Condition coverage criteria :** If all the conditions are evaluated to both TRUE and FALSE by some test set T, then the test set T satisfies Condition-Coverage criteria.
- **Full-Predicate Coverage Criterion :** If for every clause in each condition for a sequence diagram, there exists separate test case that evaluate the clause to be False and True separately, then the test set satisfies full-predicate coverage criteria.

2.8 Fault Model :

Our test strategy of generating test scenarios from UML 2.0 sequence diagrams aims to detect the fallowing type of faults.

- **Interaction Fault :** In object-oriented approach messages are passed between object to accomplish some task. These message passing between objects can create several faults such as passing a message to incorrect object, incorrect response for a message, message passing with incorrect or invalid arguments.
- **Scenario Faults :** Sequence diagram represents several scenarios for different operations. Due to some incorrect evaluation of conditions a scenario may not fallow correct sequence of messages for a particular operation.
- **Synchronization fault :** Some times there need of executing use cases in a particular order, for example execution of use case A must precede use case B and C. This fault occur in a sequence diagram when some object sends messages before completion of executing preceded messages.
- **Dependency Fault :** If a use case starts executing without satisfying the desired dependency relation then this type of fault known as dependency fault. For example in a ATM system the login use case must executed before the withdraw use case.

2.9 Test case prioritization

The purpose of test prioritization is to reduce the set of test cases based on some rational, non-arbitrary criteria, while aiming to select most appropriate tests. The size of test cases generated for a large, complex system may be very large, to execute the system for every test case in the set will be very costly and time consuming, and may also be not beneficial because it may be so happen that group of test cases are redundant and they detect the

same error.

Also the presence of bugs in some parts causes more severe and frequent failures than other parts. Test prioritization focuses on finding an ordering of the test cases such that execution of the test cases in that order meets a given criterion. So, the aim of test case prioritization is to identify critical parts of software, for which more exhaustive testing is to be carried out.

Model-based test case prioritization: In model-based test case prioritization techniques [6], the system?s model is used to prioritize test cases. Model-based test case prioritization techniques are based on software design models. If test case generation and prioritization is done early in the development life cycle, it is possible to concentrate on the effort of designing, coding, testing and maintenance activities to increases the quality of the software developed. Early prioritization of scenarios help in better planning of the design, coding, testing and maintenance activities.

2.10 XMI

XMI (XML Metadata Interchange), is an extension of XML that facilitates the standardized way for interchanging object models and metadata. Specifically, XMI is useful to programmers using the Unified Modeling Language (UML) with various languages and development tools to exchange their data models with each other.

Chapter 3

Literature Review :

UML diagrams are most widely used to model the system in the design phase. Several research attempts are made to generating and prioritizing test scenarios. Most of these work translated the control flow information presents in the UML Sequence diagrams into some other description such as a Graph. In this section we discuss some existing literature available that are closely related to our work. We divided our survey into two parts first Works related to generation of test scenarios and second is the prioritization of generated test scenarios.

3.1 Approaches to Test Scenario Generation

Sarma et al. [2] proposed an approach to generate test scenarios from UML sequence diagram, by converting sequence diagram into an directed graph called Sequence Diagram Graph (SDG), where a nodes in SDG represents a message in the sequence diagram and a directed edge represent control flow between the nodes. SDG is then used to generate test scenarios. Sarma et al.[2] used UML 1.x sequence diagram for their work, which did not support fragments such as alt, par, loop, break etc whereas our approach considers these fragments by using UML 2.x sequence diagram.

Cartaxo et al. [3] proposed a approach to generate test paths for mobile application using sequence diagram. They, constructed an intermediate model call Labeled Transition System (LTS) from sequence diagram, where directed edges ware used to represent control flow, expected output. Then, they have applied depth first search (DFS) algorithm to traverse the LTS model for generating test paths. However, their approach did not support fragments present in UML 2.x sequence diagram.

Khandai et al. [4] proposed another approach to generate test cases from sequence diagrams. They had constructe an intermediate graph called Concurrent Composite Graph (CCG) generated from sequence diagram,

which was a variant of activity diagram. Then they traversed the CCG by applying depth first search (DFS) and breath first search (BFS) to generate test cases. They have used BFS algorithm to explore fork and joint constructs.

Kansomkeat et al. [3] proposed a method for generating test sequences using UML state chart diagrams. They transform the state chart diagram into a flattened hierarchical structure of states called testing flow graph (TFG). TFG is then traversed from the root node to the leaf nodes to generate test cases. From TFG, they list possible event sequences which they consider as test sequences. The testing criterion they used to guide the generation of test sequences is the coverage of the states and transitions of TFG.

Ali et al. [8] proposed an approach for state-based integration testing. Their work builds an intermediate test model called SCOTEM (State COllaboration TEst Model) from UML collaboration diagram and the corresponding state charts. SCOTEM models all possible paths for object state transitions that a message sequence may trigger. SCOTEM then generates test paths based upon various coverage criteria. Their generated test cases aim to uncover state dependent interaction faults.

3.2 Approaches to Test Scenario Prioritization

Srivastava et al. [22] suggested prioritizing test cases according to the criterion of increased APFD (Average percentage of Faults detected) value. He proposed a new algorithm which is able to calculate the average number of faults found per minute by a test case and using this value sorts the test cases in decreasing order. He also determined the effectiveness of prioritized test case (high APFD value) compared to non-prioritized test case to compare (APFD value).

Korel et al. [23] proposed a new prioritization technique to prioritize the test cases by using several modelbased test case prioritization heuristics. Model-based test prioritization methods use the information about the system model and its behaviour to prioritize the test suite for system retesting. An experimental study has been conducted to investigate the effectiveness of those methods with respect to early fault detection. The results from the experiment suggest that system models improve the effectiveness of test case prioritization techniques.

Rothermel et al. [6] have described several techniques for test case prioritization and empirically examined their relative abilities to improve how quickly faults can be detected by those suites. Here more importance is

given to coverage based prioritization. The authors applied these techniques to the base version of a program rather than the modified version of a program. Hence these techniques are otherwise known as general prioritization techniques. The objective is to detect faults as early as possible so that the debugger will not sit idle.

Chapter 4

Proposed Scheme:

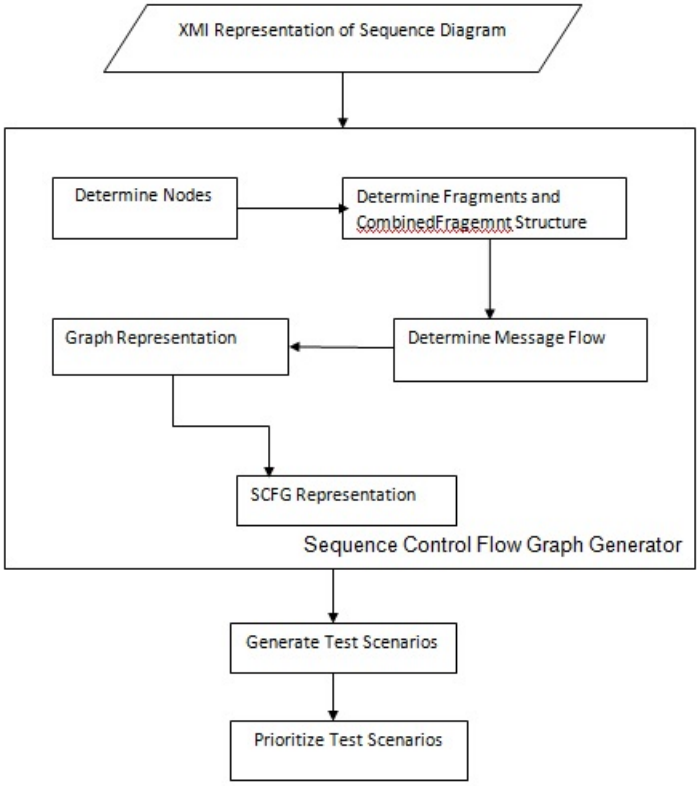


Figure 4.1: Block diagram for generating test scenarios from UML 2.x sequence diagram

In this section we discuss our approach for test scenario generation from UML 2.x sequence diagrams. Our sequence diagram includes combined Fragments using various Intraction Operators such as $alt, par, loop, break$ etc. We propose a mechanism to extract the messages in their timing

sequence and Fragments precisely from XMI representation of UML 2.x sequence diagram. Then we map every message to it's corresponding fragment. Next we generate the Sequence Control Flow Graph (SCFG) using Sequence Control Flow Generator. Then, we generate test scenarios using Test Scenario Generator .The block diagram for generating test scenarios from sequence diagram is given in Figure 3.

The major steps of our approach are partitioned into three phases.They are given below:

1. Parsing the XMI representation of UML 2.x Sequence Diagram.
2. Developing the Sequence Control Flow Graph (SCFG) generator.
3. Generating test scenarios.
4. Prioritization of text scenarios.

The first step of our approach is to parse the XMI representation of UML 2.x sequence diagram.We have used IBM Rational Rose Architecture(RSA) to draw the sequence diagram, and then we have exported the XMI representation for the sequence diagram, which is used as the input for our procedure. We propose a parser that parses the XMI file to extract information about messages, structure of fragments and combined fragments. Messages that are sent from any non user object to user object are identified. Using all the extracted information, a Sequence Control Flow Graph is generated.

In order to generate test scenarios for the sequence diagram, the SCFG needs to be traversed. We propose a Sequence Test Scenario Generation Algorithm (STSGA) for generating test scenarios. The detailed algorithm is shown in Algorithm 1. This algorithm traverses the Sequence Control Flow Graph(SCFG) to first identify the the messages that are .rom non user object to user object. Then it generates the test paths from start node to the nodes connected to these message and then finally generates test scenarios for these messages.

4.1 Sequence Control Flow Graph (SCFG)

In order to examine and visualize the control flow information present in the UML sequence diagram, we first extract all the control flow information from the XMI equivalent of the sequence diagram, then we construct an intermediate control flow graph in a testable form called Sequence Control Flow Graph (SCFG). As UML sequence diagram contains information about objects of a system in form of messages in a time sequence and we focus on functional testing of the system, we will not give emphasis

on messages sent between internal objects. For each message that is sent from internal objects to user object, our goal is to generate test scenarios taking these messages as end points. So, we will obtain SCFG, where nodes represent messages in sequence diagram and edges represent path between nodes. The messages sent from internal objects to user object are colored gray. We have added some additional nodes for the sake of simplicity in the process of generating test scenarios.

The fallowing are the type of nodes considered for constructing Sequence Control Flow Graph (SCFG).

Definition : An Sequence Control Flow Graph is a tuple
$R = \{R, M, F_{start}, F_{end}, E_{output}, C, E\}$ where,

- R is the *root node* of the Sequence Control Flow Graph(SCFG).
- M is a *message node* that represents a message from UML sequence diagram.
- F_{start} ($Fragment\ start$)is a set of nodes representing the starting of a fragment.
- F_{end} ($Fragment\ end$) is a set of nodes representing the End of a fragment.
- E_{output} ($Expected\ output$) is the set of nodes that precedes the message from internal object to user object in Sequence Control Flow Graph(SCFG).
- C ($Condition\ node$)is the set of nodes representing conditions for the fragments.
- E is the set of final nodes representing an exit of Sequence Control Flow Graph (SCFG).

4.2 Constructing the Sequence Control Flow Graph (SCFG)

We construct the SCFG for representing control flow among messages in presence of fragments and nested fragments. In this process, we give more emphasis on use scenarios [**?**] (actions executed by the user and actions viewed by the user). Each message present in the sequence diagram is represented by a node in SCFG. The start and the end of every fragment is denoted by two additional $fragment$ nodes representing starting and ending of fragment such as alt_start, alt_end, par_start, par_end etc. In alt fragment, the conditions for control flow are also denoted by additional *control* nodes containing a condition sequence number and condition itself

such as *condition*1_*true*, *condition*2_*false*, etc. The steps to build a SCFG from a sequence diagram are presented as fallows.

- The root node of SCFG is represented by a node *start*.
- The end points of SCFG are represented by the node *end*.
- From the root node *start*, for each message in the sequence diagram, a new node is added into the SCFG with it's value same as message name in the sequence diagram.
- For each message (in order it appearing sequence diagram) do the following :

  1. If the message is from user object to non-user object (internal object) or from non-user object to non-user object then a new node is added into SCFG with a directed edge from its previous node to itself.
  2. If the message is from non-user object to user object then two nodes are added into SCFG (1) first node with value *expected_output* and a directed edge from its previos node to itself. (2) second a gray color node with value same as message name and a directed edge from *expected_output* node to itself.

  Figure 1 shows an example UML 2.x sequence diagram where message passing occurs between user object and various internal objects of the system. The corresponding SCFG for Figure 1 is given in figure 2. We can observe from the SCFG in Figure 2, that all the messages of sequence diagram are represented as nodes and the starting of *alt* fragment is represented using a *Fragment start* node alt start1 and ending using a *Fragment end* node alt end1. Both the conditions for alt fragment are represented by *Condition* nodes Condition 1 True, Condition 2 False. The gray colored nodes represent messages that are passed from the internal objects to the user objects.

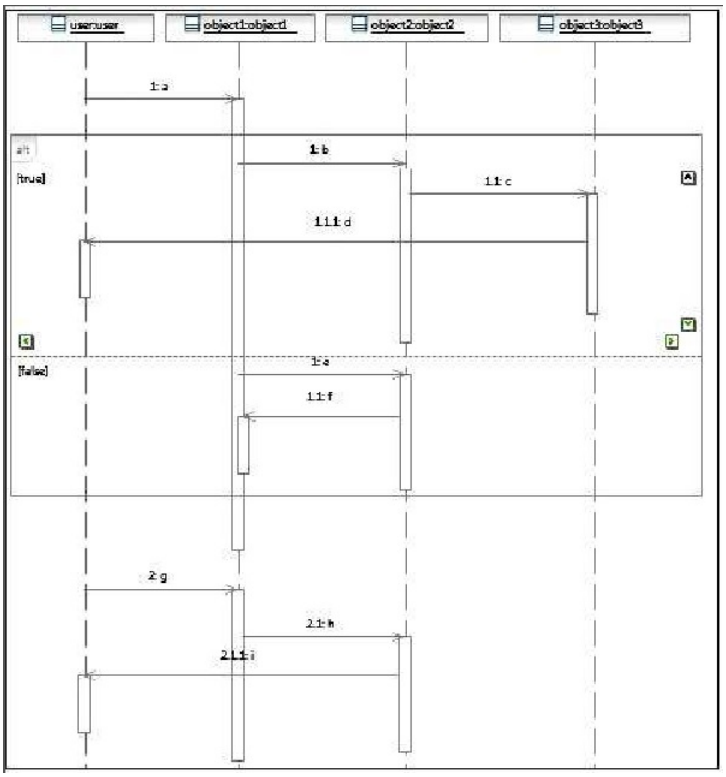


Figure 4.2: A sample Sequence diagram

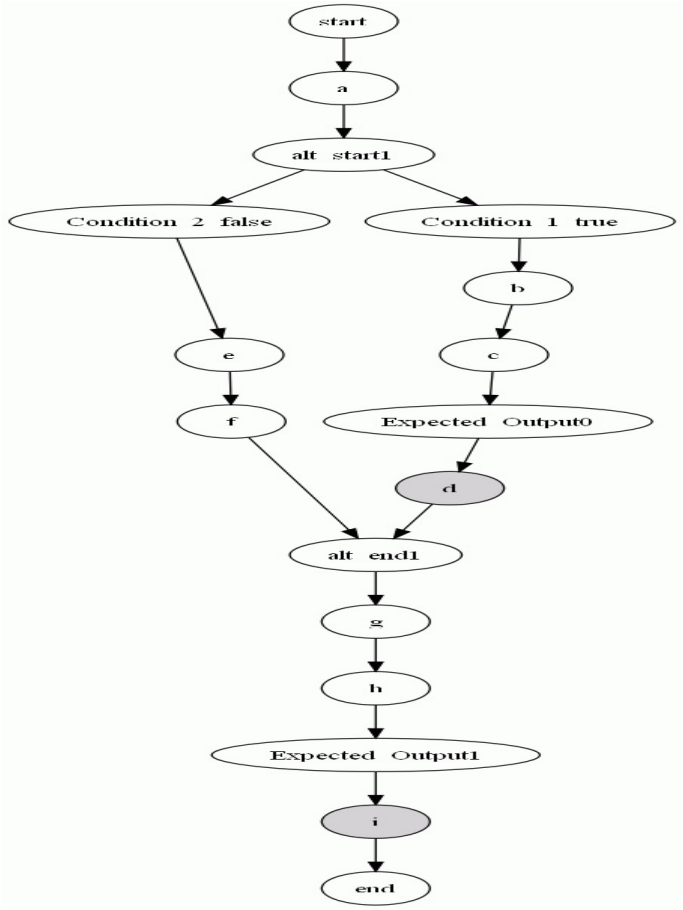


Figure 4.3: Derived SCFG from UML sequence diagram given in Figure 1

Sequence Test Scenario Generation Algorithm (STSGA).

Input: Sequence Control Flow Graph (SCFG).
Output: Set of test scenario.
runningStack=ϕ
decisionStack=ϕ
userMessageStack=ϕ
resultStack=SCFG.rootNode
for all nodes of SCFG **do**
  while SCFG.node==expextedOutput.node **do**
    userMessageStack.push(child node of SCFG.node)
  end while
end for
for all elements of userMessageStack **do**
  repeat
    if runningStack[top] $\neq$ alt.node || loop.node|| par.node|| break.node || SCFG.EndNode **then**
      resultStack.push(runningStack.pop)========== {push runningStack top element in resultStack and pop the top element from runningStack.}

```
16:         resultStack.push(child node of resultStack[top] in SCFG) {push
            child node of resultStack top element into resultStack.}
17:       else if runningStack.[top] == SCFG.EndNode then
18:         Mark the last decision node in resultStack as visitd
19:         while resultStack[top] ≠ decisionStack[top] do
20:           resultStack.pop {pop the top element fom resultStack.}
21:         end while
22:         decisionStack.pop {Pop the top element from decisionStack.}
23:       else if runningStack[top] == alt.node || break.node || loop.node
          then
24:         decisionStack.push(runningStack.pop) {pop top element of runningStack
            and push it into decisionStack and resultStack.}
25:         for all Child nodes of resultStack[top] in SCFG do
26:           if Child node is not Marked Visited then
27:             runningStack.push(Child Node) {push all child nodes of
                resultstack[top] in SCFG, if marked as visited insert into
                runningStack.}
28:           end if
29:         end for
30:       else if runningStack[top] == par.node  then
31:         resultStack.push(resultStack[top] and all its child nodes in SCFG)

32:         runningStack.push(child nodes of resultStack[top] in SCFG)
33:       else if runningStack.top == UserMessageStack.CurrentNode then

34:         resultStack.push(runningStack.pop) {Pop the top element from
            runningStack and push it into resultStack.}
35:         Print resultStack
36:         if decisionStack ≠ φ then
37:           while resultStack[top] ≠ decisionStack[top] do
38:             resultStack.pop {pop the top element fom resultStack.}
39:           end while
40:         end if
41:         if decisionStack ≠ φ then
42:           decisionStack.pop {pop the top element from decisionStack.}
43:         end if
44:       end if
45:     until runntingStack ≠ φ & decisionStack ≠ φ
46:     delete all elements from resultStack
47:     delete all elements from decisionStack
48:     delete all elements from runningStack
49: end for
50: Exit
```

4.3 Implementation Of Our Algorithm

In this section,we exemplify our approach for generating test scenarios form XMI representation of UML sequence diagram by converting XMI representation of UML sequence diagram into an equivalent Sequence Control Flow Graph (SCFG) and then generating test scenarios. We generate test scenarios from UML sequence diagram to test the feasibility and concurrency errors.

We have developed a prototype tool called XMI2SCFG (XMI to Sequence Control Flow Graph) for generating test scenarios. XMI2SCFG works in two steps (1) Parsing of XMI representation of UML Sequence diagram.(2) Creating a SCFG (Sequence Control Flow Graph) in image format, and generating test scenarios from SCFG. We have implemented XMI2SCFG in Java language using Netbeans IDE 7.0.1. XMI2SCFG takes the XMI representation of UML 2.x sequence diagram as input. We have used IBM Rational Software Architecture (RSA) 7.0 to draw the sequence diagram and then exported the XMI representation (XMI equivalent of UML sequence diagram).

In the first phase, XMI2SCFG uses SAX parser to parse the XMI representation of sequence diagram. Along with the main class and sequenceParser some auxilary classes such as *listMsg, listAlt, listPar, listLoop, listBreak* are used for this propose. The sequenceParser class implements various methods such as *getMsgID(), getMsgName(), getUserMsg(), getAltMsg(), getBreakMsg(), getParMsg(), getLoopMsg()* to interact with SAX parser. These methods process the tagged elements present in XMI representation of UML sequence diagram such as *"packagedElement", "message", "fragment", "ownedAttribute", "lifeline", "operand", "guard", "body", "ownedOperation"* to extract various information like message name, message flow, message dependacies, message type, guard condition, etc.

In the second phase,the task of XMI2SCFG is to visualize the SCFG (Sequence Control Flow Graph) in an image format. For SCFG visualization two main classes are used : DotTransformer and Graphvizvisualization. We have used for Graphviz. Taking two linked lists tranSource and tranDestination as input, the DotTransFormer objet creates a .dot file . After the .dot file is created, the methods present in graphiviz getDotSource(), getGraph(), and writeGraphTofile() create an image for SCFG (Sequence Control Flow Graph). Then the SCFG is supplied as input to Sequence Test Scenario Generation (STSG) Algorithm which generates the test scenarios. Starting from the root node *start*, STSGA scans each node of the SCFG, depending on the node type such as *message node, Fragment node, Condition node*, etc each node is processed differently in STSGA. Finally STSGA generates a set of test scenarios for UML sequence diagram.

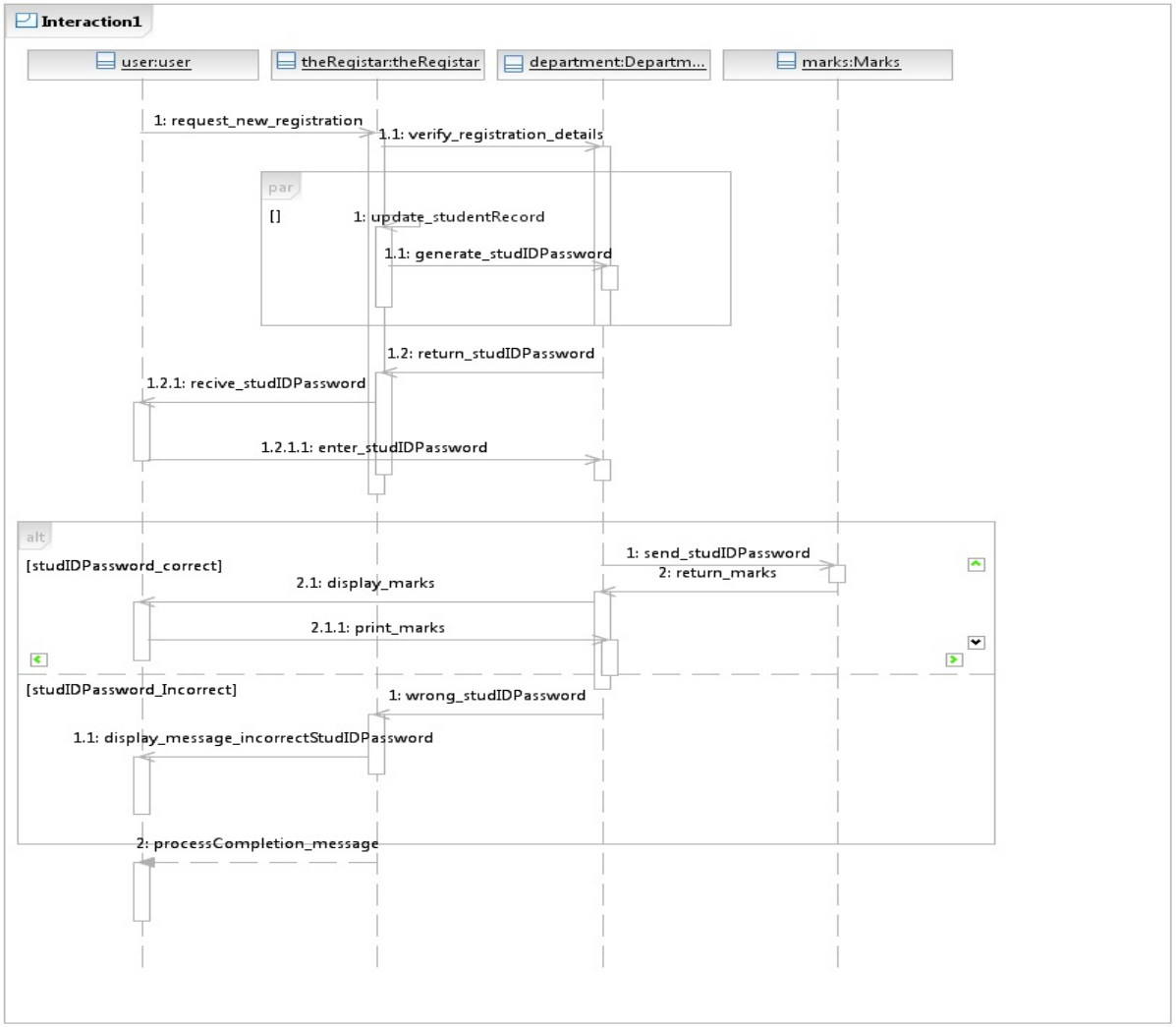


Figure 4.4: Sequence Diagram of View Marks use case

4.4 CASE STUDY

. In this section, we illustrate the working of our approach for generating test scenarios with the help of a case study pertaining to College Automation System (CAS). The CAS automates various functionalities of a college such as enrollment of students, register student, update attendance, view mark, generate receipt etc. We consider a particular use case, namely, view mark use case, a student first needs to get registered in order to view his/her marks. Once the student registered he will be provided with a student Id

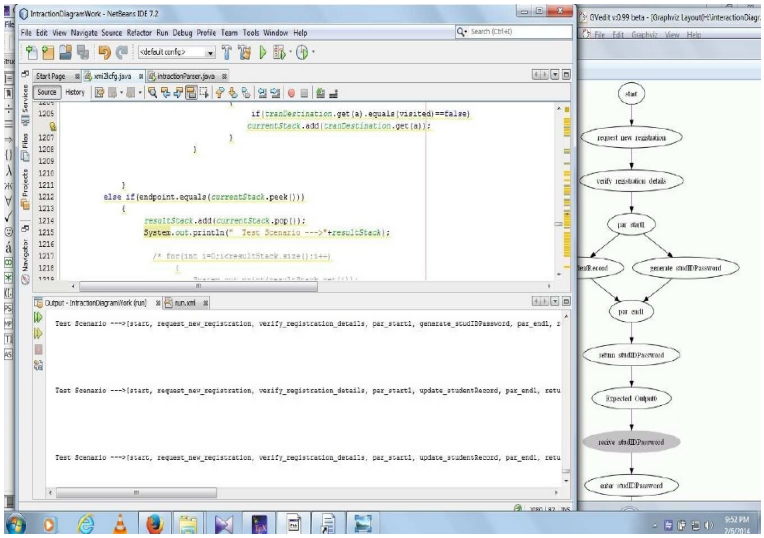


Figure 4.5: A portion of the SCFG for view mark usecase and test scenarios using XMI2SCFG

and password. By using the student ID and password, a student can get his marks.

The sequence diagram of view marks is given in Figure 4. The sequence diagram for view mark use case contains one *alt* (alternate) fragment and one *par* (parallel) fragment. The messages *recive_StudentIDPassword, display_marks, displar_message_IncorrectStudentIDPassword and processComplition_message* are the messages the user object receives from the internal objects. The complete Sequence Control Flow Graph (SCFG) generated for the view mark usecase of the sequence diagram in Figure 4 is given in Figure 6. Each message that the user object receives from internal objects is represented by gray colored nodes preceded by expected_output nodes.

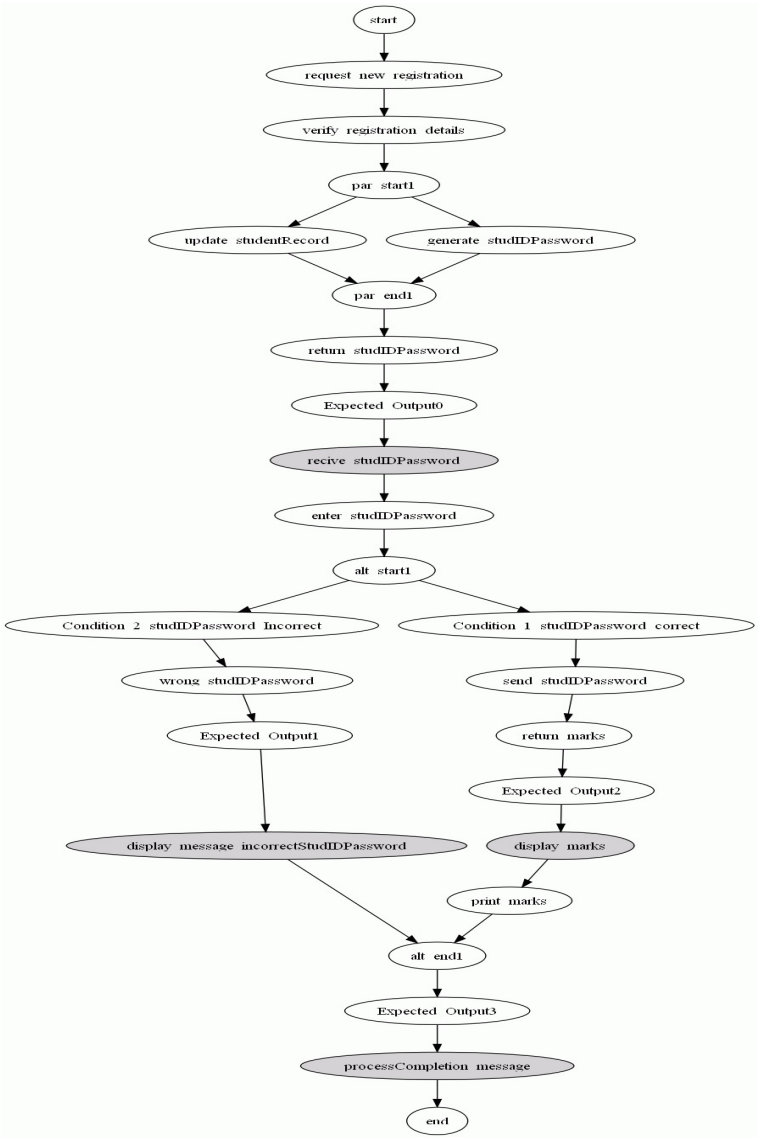


Figure 4.6: The complete SCFG of View Marks usecase of the sequence diagram given in figure 4

Table 1 shows the test scenarios are which obtained by supplying SCFG as input to our STSG algorithm

Table 4.1: Test scenarios generated for View Marks use case.

Test Scenario ID	Messages from non user object to user object	Test Scenario
TS_1	recive_studIDPassword	start → request_new_registration → verify_registration_details → par_start1 → update_studentRecord → generate_studIDPassword → par_end1 → return_studIDPassword → Expected_Output0 → recive_studIDPassword
TS_2	recive_studIDPassword	start → request_new_registration → verify_registration_details → par_start1 → generate_studIDPassword → update_studentRecord → par_end1 → return_studIDPassword → Expected_Output0 → recive_studIDPassword
TS_3	display_message_incorrect-StudIDPassword	start → request_new_registration → verify_registration_details → par_start1 → update_studentRecord → generate_studIDPassword → par_end1 → return_studIDPassword → Expected_Output0 → recive_studIDPassword → enter_studIDPassword → alt_start1 → Condition_2_studIDPassword_Incorrect → wrong_studIDPassword → Expected_Output1 → display_message_incorrectStudIDPassword
TS_4	display_message_incorrect-StudIDPassword	start → request_new_registration → verify_registration_details → par_start1 → generate_studIDPassword → update_studentRecord → par_end1 → return_studIDPassword → Expected_Output0 → recive_studIDPassword → enter_studIDPassword → alt_start1 → Condition_2_studIDPassword_Incorrect → wrong_studIDPassword → Expected_Output1 → display_message_incorrectStudIDPassword
TS_5	display_marks	start → request_new_registration → verify_registration_details → par_start1 → update_studentRecord → generate_studIDPassword → par_end1 → return_studIDPassword → Expected_Output0 → recive_studIDPassword → enter_studIDPassword → alt_start1 → Condition_1_studIDPassword_correct → send_studIDPassword → return_marks → Expected_Output2 → display_marks
TS_6	display_marks	start → request_new_registration → verify_registration_details → par_start1 → generate_studIDPassword → update_studentRecord → par_end1 → return_studIDPassword → Expected_Output0 → recive_studIDPassword → enter_studIDPassword → alt_start1 → Condition_1_studIDPassword_correct → send_studIDPassword → return_marks → Expected_Output2 → display_marks
TS_7	processCompletion_message	start → request_new_registration → verify_registration_details → par_start1 → update_studentRecord → generate_studIDPassword → par_end1 → return_studIDPassword → Expected_Output0 → recive_studIDPassword → enter_studIDPassword → alt_start1 → Condition_1_studIDPassword_correct → send_studIDPassword → return_marks → Expected_Output2 → display_marks → -> print_marks → alt_end1 → Expected_Output3 → processCompletion_message
TS_8	processCompletion_message	start → request_new_registration → verify_registration_details → par_start1 → generate_studIDPassword → update_studentRecord → par_end1 → return_studIDPassword → Expected_Output0 → recive_studIDPassword → enter_studIDPassword → alt_start1 → Condition_1_studIDPassword_correct → send_studIDPassword → return_marks → Expected_Output2 → display_marks → -> print_marks → alt_end1 → Expected_Output3 → processCompletion_message
TS_9	processCompletion_message	start → request_new_registration → verify_registration_details → par_start1 → update_studentRecord → generate_studIDPassword → par_end1 → return_studIDPassword → Expected_Output0 → recive_studIDPassword → enter_studIDPassword → alt_start1 → Condition_2_studIDPassword_Incorrect → wrong_studIDPassword → Expected_Output1 → display_message_incorrectStudIDPassword → alt_end1 → Expected_Output3 → processCompletion_message
TS_{10}	processCompletion_message	start → request_new_registration → verify_registration_details → par_start1 → generate_studIDPassword → update_studentRecord → par_end1 → return_studIDPassword → Expected_Output0 → recive_studIDPassword → enter_studIDPassword → alt_start1 → Condition_2_studIDPassword_Incorrect → wrong_studIDPassword → Expected_Output1 → display_message_incorrectStudIDPassword → alt_end1 → Expected_Output3 → processCompletion_message

Chapter 5

Test Scenario Prioritization

Now-a-days, complexity and the size of the software is rapidly increasing because the of object-oriented approach. To test these complex and large systems we need a efficient test suite. Exhaustive testing of these large systems in impractical and costly. There are certain critical pats of the system, where the chance of error occurrence is more. Prioritization aims to detect these critical errors and to reduce the size of the test in a efficient manner, so the time and the cost involved in testing can be reduced. In our proposed approach for test scenario prioritization, the SCFG obtained from the sequence diagram is used. It may be noted that each path from start node to end node, correspond to a scenario of the use case. Steps of our proposed priotization technique are :

- Assign weights to the nodes of the SCFG.
- Assign weights to the edges of the SCFG.
- Calculate the weight of each path (Scenario).
- Prioritize Scenarios.

1. **Assign weights to the nodes of the SCFG :**
   Weights are assigned to nodes based on their complexity, the node which represent a fragment (such as alt_start, par_start etc) in the intermediate graph are assigned with the highest weight of two, because they represent decision in the sequence diagram and chance of defect occurrence in more in these nodes. The rest of the nodes are assigned with the weight one. Hence we assign wiegt to a node *"n"* as fallows :

$$Weight(n) = \begin{cases} 2 & \text{for fragment start nodes} \\ 1 & \text{if message nodes} \end{cases}$$

2. **Assign weights to the edges.**
   The weight for a edge *"e"* is assigned as fallows:

$$Weight(e) = (n_i)_{in} \times (n_j)_{out}$$

   Where $(n_i)_{in}$ is the number of incoming edges of node n_i and $(n_j)_{out}$ is the number of outgoing edges of node n_j and e is the edge connecting n_i and n_j.

3. **Calculate the weight of each path.**
   Total weight for a path *"p"* is the sum of weights of all the nodes and edges allong the path *"p"*. So we calculate the path weight as fallows:

$$Weight(p) = \sum_{i=1}^{n} Weight(n_i) + \sum_{j=1}^{m} Weight(e_j)$$

   Where n represent node, e represent edge and n, m is the number of nodes and number of edges receptively along the path p.

4. **Prioritize Scenarios :**
   Considering the weights of each path, we prioritize the scenarios for the corresponding paths in order of decreasing weights.

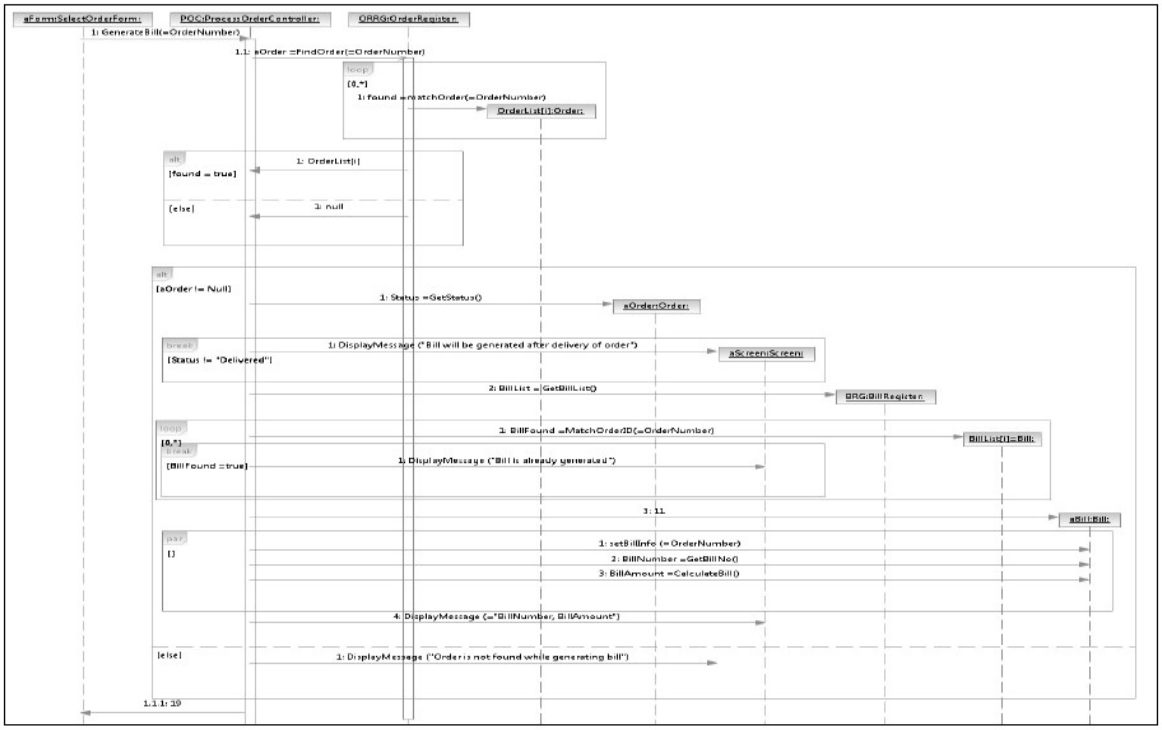


Figure 5.1: UML 2.0 Sequence Diagram of Generate Bill

5.1 Working of the proposed test scenario prioritization approach

We discuss our proposed approach for test scenario prioritazation with the help of a case study pertaining to a Restaurant Management System. The RAS automates various functionalities of a restaurant such as Make Order, Process Order and Generate Bill etc. Here, we take only on a particular use case, namely, Generate Bill. In Generate Bill use case, manager of the restaurant inputs Order Number of an order whose Bill is to be generated. Depending on the current status of the order and whether Bill has already been generated for this Order or not, many scenarios can occur, which are modeled in the sequence diagram as shown in Fig. 5.1. The intermediate control flow graph obtained from the *Generate Bill* use case sequence diagram, by using our tool XMI2SCFG discussed in section 4.3, is shown in Figure 5.2. The set of test scenarios generated from the intermediate control flow graph of the *Generate Bill* use case is shown in Table 5.1 .

Table 5.1: Generated Test Scenarios from Generate Bill Sequence Diagram

	Scenario or path sequence
1	Start->M1 ->M2 ->loopS1->M3->loopE1->loopS1->altS1->M4->altE1 ->altS2 ->M6 -> breakS1 ->M7-> breakE1 ->end
2	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M6-> breakS1->M7-> breakE1->end
3	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M7->breakE1->end
4	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M7->breakE1->end
5	Startt->M1t->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2-> M9-> breakS2->M10->breakE2->end
6	Start->M1->M2->loopS1->M3->loopE1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9-> breakS2-> M10->breakE2->end
7	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2->M10-> breakE2-> end
8	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2->M10-> breakE2-> end
9	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2->M11->parS1->M12->M13->M14->parE1->M15->M16-> altE2->M18-> end
10	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2->M11->parS1->M12->M13->M14->parE1->M15->M16->altE2-> M18-> end
11	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M12->M13-> M14->parE1->M15->M16->altE2->M18->end
12	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M12->M13-> M14->parE1->M15->M16->altE2->M18->end
13	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M11->parS1->M12->M13->M14->parE1->M15->M16->altE2->M18->end
14	Start->M1->M2->loopS1->M3->loopE1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11-> parS1->M12->M13->M14->parE1->M15->M16->altE2->M18->end
15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M12-> M13->M14->parE1->M15->M16->altE2->M18->end
16	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M12->M13->M14->parE1->M15->M16->altE2->M18->end
17	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M17->altE2->M18->end
18	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M17->altE2->M18->end
19	Start->M1->M2->loopS1-> altS1->M4->altE1->altS2->M17->altE2->M18->end
20	Start->M1->M2->loopS1-> altS1->M5->altE1->altS2->M17->altE2->M18->end

After the test scenarios are generated we apply our approach to prioritize the scenarios, first we assign weight to the edges and nodes of the intermediate control flow graph by using the equation 1 and 2 respectively, intermediate

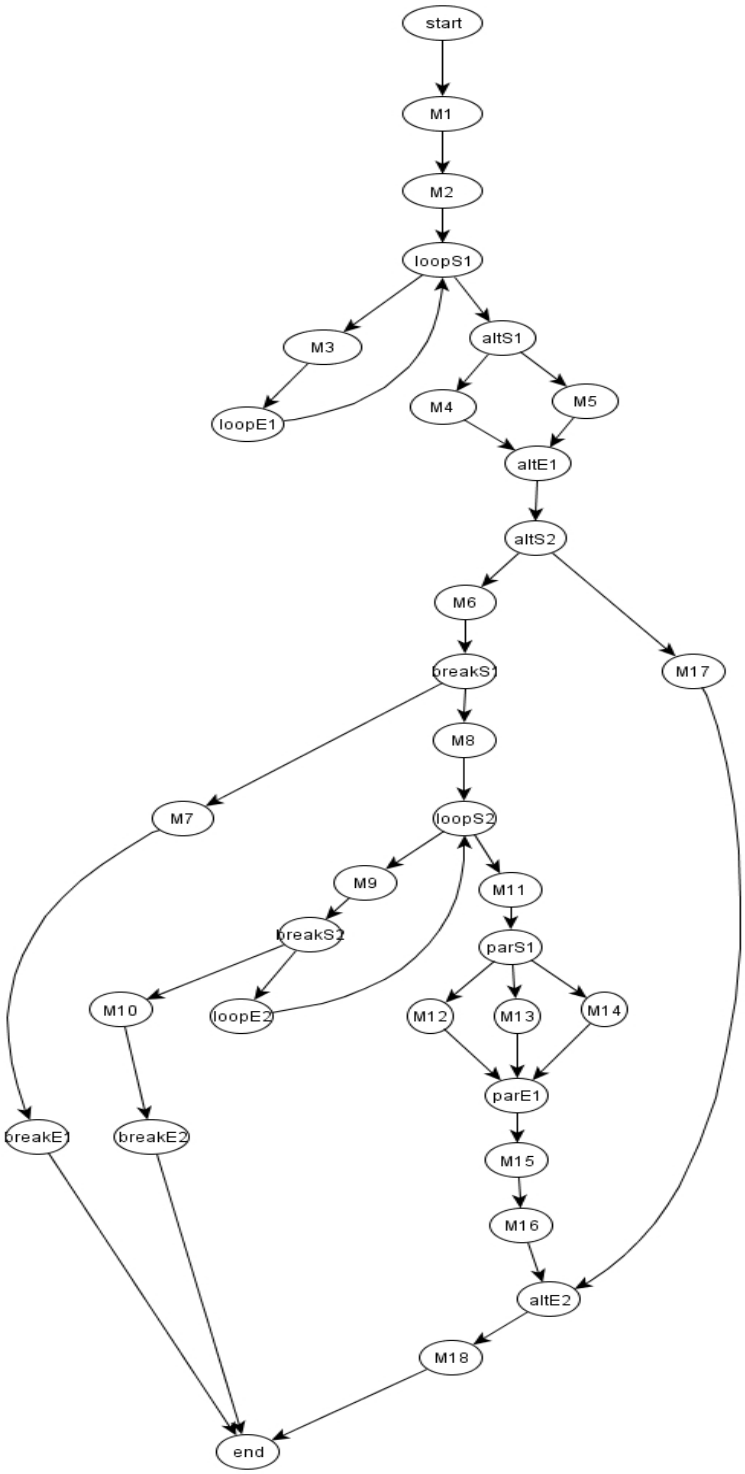


Figure 5.2: SCFG obtained from Generate Bill use case sequence diagram

control flow graph with weights assigned to edges and nodes is shown in figure 10.

In the next step we calculate the total path weight for each scenario by using equation 3, the total path weight is the sum of wieghts of nodes and path in a particular path. Test scenarios with total path weight is shown in Figure 5.3. .

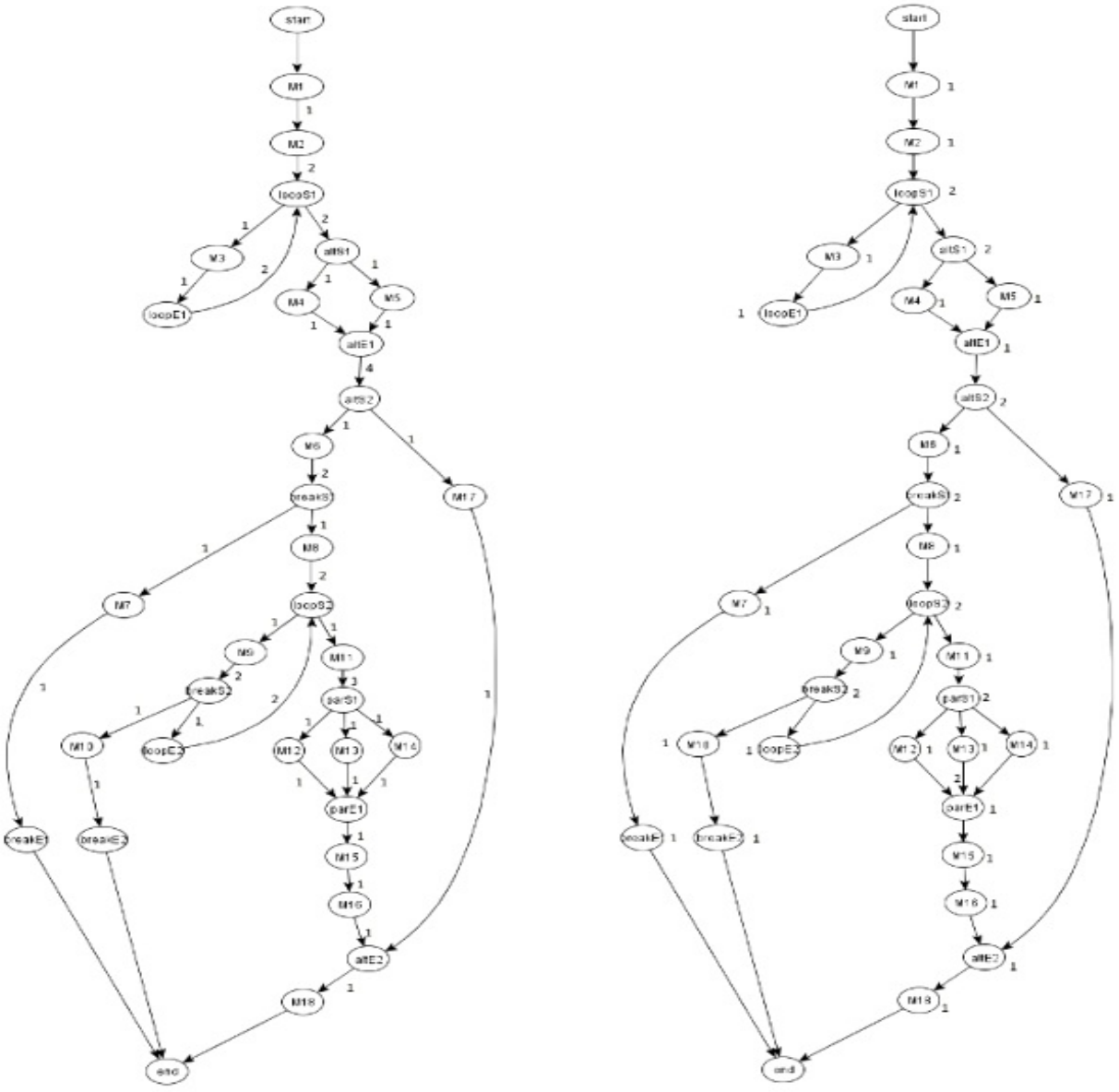


Figure 5.3: CFG after assigning weight to nodes and edges

Test scenarios along with their corresponding total node weight, total

Table 5.2: Test Scenarios with node weight, edge weight and total weight of path

TSQSA	Scenario or path sequence	Node weight	Edge weight	Total weight of path	Priority
1	Start->M1 ->M2 ->loopS1->M3->loopE1->loopS1->altS1->M4->altE1 ->altS2 ->M6 -> breakS1->M7-> breakE1 ->end	17	20	37	13
2	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M6-> breakS1->M7-> breakE1->end	17	20	37	14
3	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M7->breakE1->end	14	16	30	18
4	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M7->breakE1->end	14	18	32	17
5	Startt->M1t->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2-> M9-> breakS2->M10->breakE2-> end	25	28	53	9
6	Start->M1->M2->loopS1->M3->loopE1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9-> breakS2-> M10->breakE2->end	25	26	51	10
7	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2->M10->breakE2->end	21	23	43	12
8	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2->M10->breakE2->end	21	23	44	11
9	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M9->breakS2->loopE2->loopS2-> M11->parS1->M12->M13->M14->parE1->M15->M16->altE2->M18->end	38	39	77	1
10	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8-> loopS2->M9->breakS2->loopE2->loopS2-> M11->parS1->M12->M13->M14->parE1->M15->M16->altE2-> M18->end	37	39	76	2
11	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M12->M13->M14->parE1->M15->M16->altE2->M18->end	33	35	68	4
12	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M12->M13->M14->parE1->M15->M16->altE2->M18->end	33	37	70	3
13	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M11->parS1->M12->M13->M14-> parE1->M15->M16->altE2->M18->end	31	33	64	6
14	Start->M1->M2->loopS1->M3->loopE1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11-> parS1->M12->M13->M14->parE1-> M15->M16->altE2->M18->end	31	34	65	5
15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M12-> M13->M14->parE1->M15->M16-> altE2->M18->end	27	28	55	8
16	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M12->M13->M14->parE1->M15->M16-> altE2->M18->end	27	29	56	7
17	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M17->altE2->M18->end	17	18	35	16
18	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M17->altE2->M18->end	17	19	36	15
19	Start->M1->M2->loopS1-> altS1->M4->altE1->altS2->M17->altE2->M18->end	13	14	26	20
20	Start->M1->M2->loopS1-> altS1->M5->altE1->altS2->M17->altE2->M18->end	13	14	27	19

edge weight, total path weight and priority's is shown in Table 5.2. The priorities are assigned to the scenarios in a decreasing order of their total path weight. Hence we have two possible test scenario prioritized order shown in Table 5.3. We have selected the first order as our test scenario prioritized order.

Table 5.3: Prioritized Order

1	TS_9,TS_{10},TS_{12}, TS_{11},TS_{14},TS_{13},TS_{16},TS_{15}, TS_5, TS_6, TS_8,TS_7,TS_1,TS_2,TS_{18}, TS_{17},TS_4,TS_3,TS_{20},TS_{19}
2	TS_9, TS_{10}, TS_{12}, TS_{11}, TS_{14}, TS_{13}, TS_{16}, TS_{15}, TS_5, TS_6, TS_8, TS_7, TS_2, TS_1, TS_{18}, TS_{17}, TS_4, TS_3, TS_{20}, TS_{19}

5.2 Analysis of test scenario prioritization technique

To measure the effectiveness of our prioritized test scenarios, we have used ModelJunit to design a test suite and written tests for all the test scenarios we have generated earlier. We have executed the test by including a total of fifteen faults (Randomly we have taken) in to the test cases. We execute the test by introducing one faults at a time and observed which are the test scenarios are able to detect the fault, this process is repeated for all fifteen faults. Figure 5.4 shows a screenshot of test scenarios with fault detected using ModelJUnit.

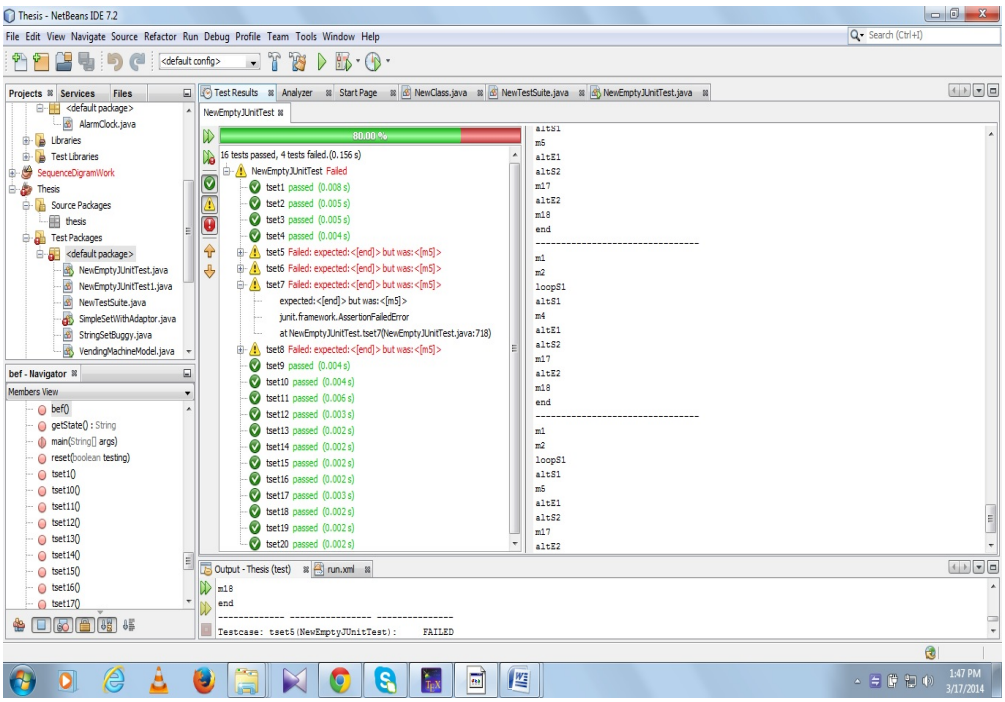


Figure 5.4: Test scenarios with fault detected screenshoot

Table 5.4: Test scenarios with fault detected

F/TS	T_1	T_2	T_3	S_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}	T_{16}	T_{17}	T_{18}	T_{19}	T_{20}
F_1									×	×	×	×								
F_2	×	×	×	×																
F_3						×	×	×	×		×	×								
F_4					×	×	×	×												
F_5									×	×	×	×	×	×	×	×				
F_6																		×	×	×
F_7									×	×	×	×	×	×	×	×				
F_8					×	×	×	×	×	×	×	×								
F_9		×		×					×	×								×		×
F_{10}	×	×											×	×	×	×				
F_{11}					×				×	×							×			
F_{12}									×	×					×	×				
F_{13}			×	×						×	×		×	×			×	×		
F_{14}					×	×			×	×	×	×		×	×				×	
F_{15}	×	×	×						×							×	×			
Total	3	4	2	2	4	4	4	4	10	9	7	6	4	5	5	5	3	3	2	2
Time [in ms]	4	5	4	4	3	4	3	3	7	6	3	2	3	3	2	2	3	3	2	2

5.2.1 Average Percentage of Fault Detected (APFD):

Average Percentage of Fault Detected (APFD) measures the weighted average of the percentage of faults detected over the life of the test suite. Higher

value of APFD imply faster of better fault detection rate. According to [25], the APFD value for a test suite can be calculated by Eq 5.1

$$APFD = 1 - ((T_{F1} + T_{F2} + ... + T_{Fm})/n * m) + 1/2 * n \quad (5.1)$$

Where T represent the test suite, m is the number of faults in the program, n is the number of test scenarios and T_{Fi} is the position of the first test scenario in T that detects the fault i.

Table 5.4 shows the number of faults detected by each test scenario in the test suite against all fifteen faults. There are twenty number of test scenarios in the test suite. So the number of faults (m) = 15 and number of test scenarios (n) = 20.

Now let us compute the APFD value by applying Eq 5.1 to the prioritized test scenarios. Hence Putting the values of m, n, T_{Fi} (The position of the first test scenario in the ordering T' of T that exposes fault i) in the APFD Eq. 1 we get

APFD (*Prioritized test scenario*) = 1-((1 + 1 + 2 + 2 + 1 + 1 + 4 + 8 + 1 + 1 + 2 + 13 + 12 + 13 + 2 + 14 + 16 + 19 + 1 + 3) / (20 * 15)) + 1/(2 * 20) = **0.675**

The APFD value for a non-prioritized test sequence (e.g. T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}, T_{17}, T_{18}, T_{19}, T_{20}) can be calculated as fallows :

APFD (*Non-prioritizes test scenario*) = 1-((9 + 9 + 10 + 10 + 9 + 9 + 16 + 13 + 9 + 9 + 10 + 2 + 6 + 2 + 10 + 4 + 17 + 19 + 9 + 12) / (20 * 15)) + 1/(2 * 20) = **0.385**

It may be observed that value of APFD for prioritized test scenarios is higher then the value of APFD obtained for non-prioritized order. Hence the increase in APFD value justifies our approach for test scenario prioritization to be effective.

5.3 Implementation and Results:

5.3.1 Implementation:

For implementing the proposed enhanced Relative Frequency Model we used NetBeans IDE, JDK 1.6, Apache Lucene, Word-net.

when we run the executable jar file, we will get an user interface to upload test and registered documents. We can view the user interface for uploading the documents in Fig. 7.

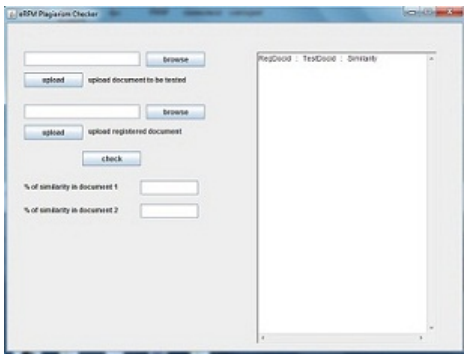


Figure 5.5: eRFM interface

Then we will use a file chooser to select the respective document(Fig. 8). After that we need to perform indexing on both the selected documents(Fig. 9).

Table 5.5: execution times for eRFM and SCAM

Method/no of documents	50	100	150	200	250	300
SCAM	1987	4190	6067	7975	9887	11927
eRFM	1071	2210	3205	4185	5207	6179

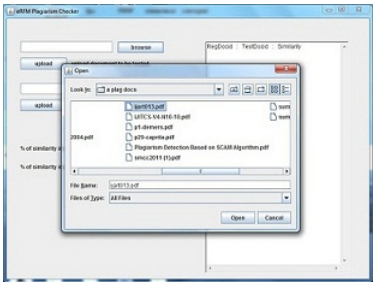


Figure 5.6: File Chooser for selecting documents

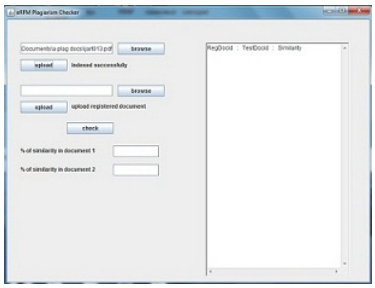


Figure 5.7: Indexed Successfully first document

5.3.2 Results and Comparisons:

We can observe the % similarity between the both documents in Fig. 10 as well as the matching sentence numbers of registered document and test documents. And we can observe the execution times take by the SCAM and eRFM in Table 1.

And we plotted a graph(Fig. 11) between number of documents and time taken to execute the algorithm for both SCAM and eRFM algorithm. Here in the graph above line shows the behavior of SCAM algorithm as the number of documents are increasing, below line shows the behavior of eRFM.

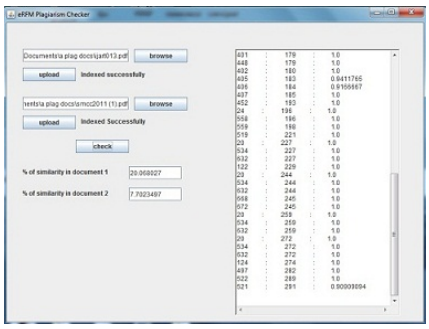


Figure 5.8: % of similarity along with similarity between test and registered document sentences

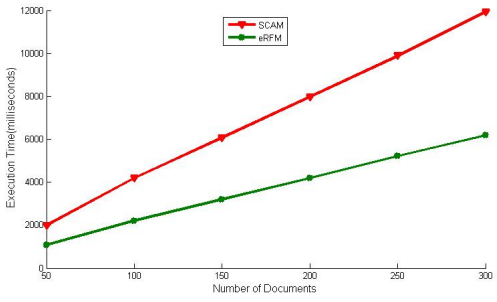
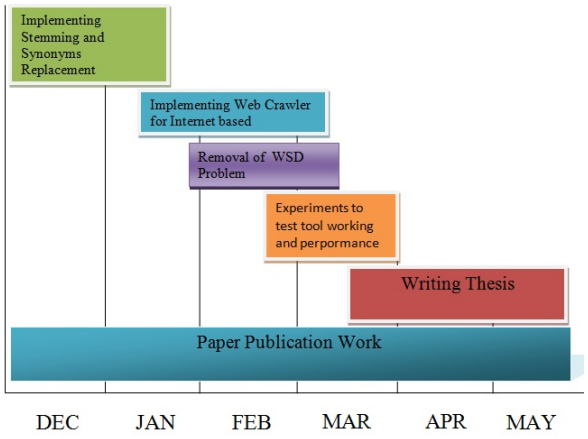


Figure 5.9: number of documents vs execution time graph

As our comparison some times against huge data sets having hundreds of documents, then the execution time is really matters. In our new approach there is no need to calculate the closeness set, which avoids the problem "selection of e value", and reduces the time required to get the results, it means gives us the results faster.

And here the comparison is sentence by sentence, where usage of a whole sentence as a subset of test document sentence is rare. But authors some times may use one or two words of registered sentence, because of that we cant say its a plagiarism. We eliminated that problem of misleading in our approach. let us take a simple example.
Ex: regstring = a,b and teststring = a,b,e,f,g,h
SCAM gives similarity value = max((2/2),(2/6)) = 1 . it means textstring is a copy of regstring but its not exactly.
eRFM gives similarity value = 2/6 = 1/3. which less than our threshold value.

5.4 Roadmap Ahead:



5.5 Summary:

In this paper, we have proposed a novel approach for test scenario generation from UML 2.x sequence diagram considering the fragments, nesting of fragments and control flow primitives present in sequence diagrams. The method first generates an intermediate graph called Sequence Control Flow Graph (SCFG) from the XMI representation of UML 2.x sequence diagram. Then by analyzing the control flow information, message sequence and the fragment structure, our proposed approach generates test scenarios, for

various use case present in a system. Most of the existing techniques of test scenario generation from UML sequence diagrams are manual and do not consider fragments and nesting of fragments into test scenarios. Hence, these methods become more complex while taking UML 2.x sequence diagrams.

Our approach is a fully systematic interpretation of control flow information for various fragments as well as nested fragments present in UML 2.x sequence diagram. Subsequently our approach uses these control flow primitives for test scenario generation.Our approach is fully automatic. The test scenarios thus generated are suitable for functional testing and detecting interaction and scenario faults.